Taverna, reloaded

Taverna, reloaded

Paolo Missier[1], Stian Soiland-Reyes[1], Stuart Owen[1], Wei Tan[2], Alexandra Nenadic[1], Ian Dunlop[1], Alan Williams[1], Tom Oinn[3], and Carole Goble[1]

[1] School of Computer Science, The University of Manchester, UK
[2] Mathematics and Computer Science Division, Argonne National Laboratory, Argonne., IL., USA
[3] European Bioinformatics Institute, UK

Abstract. The Taverna workflow management system is an open source project with a history of widespread adoption within multiple experimental science communities, and a long-term ambition of effectively supporting the evolving need of those communities for complex, data-intensive, service-based experimental pipelines. This short paper describes how the recently overhauled technical architecture of Taverna addresses issues of efficiency, scalability, and extensibility, and presents performance results based on a collection of synthetic workflows, as well as a concrete case study involving a production workflow in the area of cancer research.

1 Introduction

Taverna [7] is a workflow language and computational model designed to support the automation of complex, service-based and data-intensive processes. Although Taverna has been successfully applied in domains as diverse as bioinformatics, astronomy, medical research, and music, it is perhaps best known for its application to the Life Sciences [4], where it has been used to support experimental investigation into a variety of research areas, including gene and protein sequence and structure annotation, proteomics, microarray analysis, text mining, systems biology, and more. The first version, launched in 2004 [13], has enjoyed broad adoption over the years[4], owing in part to a fairly intuitive model for service composition, and to a growing number (in the order of tens of thousands) of available services, mostly community-provided and free to use, and to Taverna's ability to invoke ad hoc scripts and Java object methods.

Taverna combines a dataflow model of computation, whereby a workflow consists of a set of processors (representing software components such as Web Services) that are connected through data dependencies links, with a functional model that accounts for collection-oriented processing. This hybrid model is designed to strike a balance between expressivity and simplicity, with the ultimate goal of empowering users, who may have only a rudimentary understanding of programming, to assemble complex workflows. While the model and its theoretical underpinnings [16] have remained largely stable over the years, the evolving requirements of e-science applications have recently prompted a radical re-design of the architecture, which we will refer to as "Taverna 2" (T2 for short) to distinguish it from the previous version, T1.x.

This paper presents the salient features of the new architecture[5]. The new architecture has two main goals. Firstly, to improve the scalability of workflow execution, both in terms of data volume and execution times, over its predecessor. Secondly, to provide third-party developers with clear configuration points for performance tuning, and with extensibility points for adding new high-level constructs, such as a while-loop, to the dataflow model, in order to facilitate workflow design in paradigmatic scenarios. More specifically, the following architectural requirements have inspired the design of the T2 architecture:

---

[4] In 2008 there were over 4000 active users of Taverna, for over 57,000 downloads total.

[5] More details can be found in a complete techreport, available online at `http://bit.ly/9RglCZ`.

Parallelism. Models of computation that combine dataflow and functional models are known to facilitate parallel execution [10], making the exploitation of the parallelism provided implicitly by the workflow specification a realistic goal. Potential parallelism can be found both amongst processors (*inter-processor*), by statically determining data dependencies amongst processors in the dataflow graph, as well as amongst multiple invocations of the same processors (*intra-processor* data parallelism), i.e., on individual elements of an input collection.

Configurability. Each processor may have different operational requirements, for example regarding its tolerance to transient error conditions in the underlying service invocation. Thus, it should be possible to fine-tune the behaviour of each processor independently from that of the others.

Openness. It should be possible for third party contributors to extend the functionality of the execution model in a principled way. For example, interacting with asynchronous services that require periodic polling to check on result availability, does not fit the data-driven model well. While in T1.x the model can be "stretched" to simulate polling processors, T2 accommodates this requirement by implementing a limited form of while-loop construct by exploiting a generic extensibility mechanism, called the *dispatch stack*.

Separation of data and process spaces. For data-centric computing to scale to arbitrary data volumes, the data space should be managed separately from the process space, and, whenever possible, data should be passed from one processor to the next by reference rather than by value.

This paper describes how the design principles listed above have been translated into a coherent architectural design for T2, and presents performance results for the current implementation.

Workflow modelling for data-intensive scientific applications is, of course, not new. One of the recognized challenges of scientific workflow management systems is to provide abstract modelling constructs that are then automatically instantiated as an orchestration of concrete tasks that execute on an underlying parallel architecture [2, 6]. This is the approach taken for example by the Pegasus system [3], with the goal of decoupling the logical specification of the workflow from the pool of resources required to execute it. In this paradigm, resources are allocated by the scheduler incrementally and dynamically. In a similar fashion, Chimera [5] provides a dedicated language (VDL) for the partial specification of logical workflows. A complementary, bottom up approach, is to start with a job management system, like Condotr, and then provide users with a model for building complex workflows from a pool of individual jobs [1]. Another example of concrete and well-known mechanism used for the specification of complex dataflows is *shell pipes*, proposed in [19].

In Sec. 2 we briefly present the dataflow computation model that underpins parallel and pipelined workflow computation, describe the configurable and extensible processor execution model, and present the main architectural solutions for parallelism. Sec. 3 provides performance figures for the current implementation, and finally in Sec. 4 we present the architecture in action on a concrete case study from the caGrid project.

2   Workflow processing model and architecture

We now describe some of the architectural solutions used to realize the principles enunciated in the introduction. The overview architectural diagram is shown in Fig. 1.

A Taverna workflow, described in detail in [16] is specified by a directed graph where nodes, called *processors*, represent software components, typically Web Services or local scripts. A processor node consumes data that arrives on its *input ports* and produces data on its *output ports*.
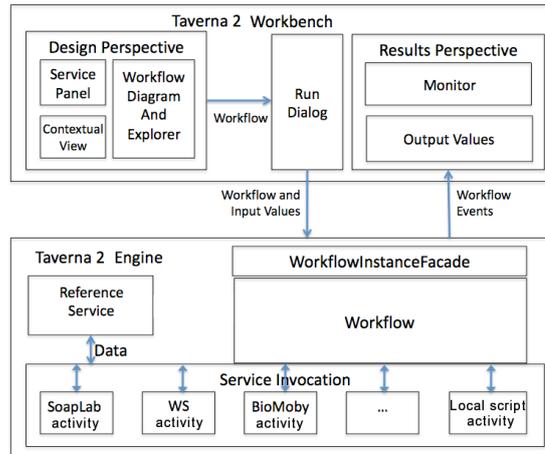
Fig. 1. Overview block diagram of Taverna 2 architecture

Each arc in the graph connects a pair of ports, and denotes a data dependency from the output port of the *source* processor to the input port of the *sink* processor. In this model, data items are either of a simple type (string, number, etc.), or are lists of items, nested to arbitrary levels.

Conceptually, a workflow computation proceeds by pushing data through the directed data links from one processor to all of its successors, starting with the items that are presented on the workflow inputs. A processor is ready to execute when all of its inputs ports are populated with a data item. A processor's execution consists of the invocation of an associated *activity*, for example a Web Service or a local script, and produces new data items on its output ports. These are then propagated along the outgoing arcs.

A workflow specification is compiled into a multi-threaded object model (implemented in Java), where processors are represented by objects, and data transfers from output to input ports of downstream processor objects are realized using local method invocations between objects. Data-driven computation is realised by mapping each processor to an object, which independently starts its own execution in a separate thread, as soon as all of its (connected) input ports are populated with a data item, and it transfers its output along the arcs upon completion.

In T2, data elements are only loaded into the execution process space on demand, when required by some processor's input port, and are swapped out again to a separate persistent storage when they are no longer needed. Thus, while the same data may be transferred back and forth from storage multiple times, the total amount of main memory required by an execution is bounded. The max memory footprint is determined by the size of each data item and the number of concurrent threads that require in-memory data at any given time. As shown in the use case presented in Sec. 4, the max number of active threads, a parameter that can be configured independently for each processor, can be used to control the total amount of memory required.

Such separation of the data from the workflow execution space is achieved by registering all values that are produced during a computation with a new Data Manager (DM). The DM's main function is to index the values by assigning to them a unique data reference (a URI) and store them into a database, from where a processor that requires the values as part of its input can retrieve them using the reference.

2.1 Configurable processing

The exact sequence of operations that occurs upon invocation of an activity associated to a processor is configurable, making for a flexible and extensible workflow execution model. The *interceptor* design pattern is used to configure each individual processor. Specifically, a *dispatch stack* consisting of an extensible set of *layers* is associated to each processor. Each layer is responsible for a particular feature, typically associated with Quality of Service. In the standard configuration, the stack consists of the following layers, as shown in Fig. 2(a):
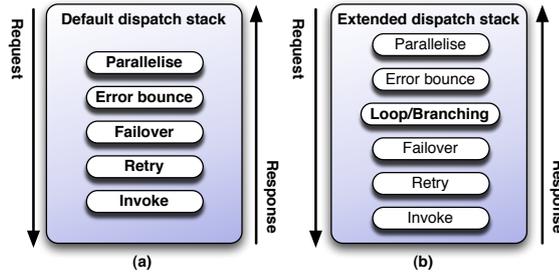
Fig. 2. Processor dispatch stack

Parallelise: this layer ensures that, when iterations over lists are involved, independent concurrent threads are created to process each list element;

Error Bounce: is responsible for immediately terminating an execution when any of the inputs are in an error state. This protects the underlying activities from being invoked on invalid input;

Failover: detects the failure of an activity associated to the processor, and is responsible for selecting an alternate activity, if available (recall that activities can be added and removed dynamically);

Retry: provides tolerance to transient errors in the underlying activity, by repeatedly attempting the same invocation for a configurable number of times.

Each of these layers exposes a set of configuration parameters to the workflow designer. The stack is activated by a request message to the processor, consisting of the data on the input ports. The message is pushed down the stack, where each layer performs its function and, if needed, forwards a new version of the request to the next layer. At the bottom of the stack, the *Invoke* layer performs the actual invocation of an activity; in the case of a Web Service invocation, for instance, this layer is a Web Service client that maps the incoming request to a SOAP message, and manages the interaction with the service. The result is mapped to a response message, which finds its way back up along the stack, where it is intercepted by each layer, possibly transformed, and forwarded. Ultimately, the processor invocation terminates and the response is forwarded to downstream processors along the data links.

In this architecture, a private instance of the dispatch stack is associated to each processor, making it configurable independently of the others. Also, the interceptor pattern naturally allows new layers to be added to the standard stack shown in Fig. 2(a). Useful additional layers that are already available as part of the current release include the *while-loop* layer, shown in Fig. 2(b), as well a *Provenance layer*, designed to generate audit events from the processor's execution as a basis for collecting workflow provenance [12].

2.2   Parallelism and pipelining

The majority of Taverna workflows, for the most part in the bioinformatics domain, are rather more data-intensive than compute-intensive. Typically, these workflows perform some form of "on-the-fly data integraton" on large collections of data values retrieved from a variety of databases through their Web service interfaces, exposed as Taverna processors. In this setting, each of the available parallel threads tends to carry a potentially large data item, such as an image, while involving a relatively small amount of computation. Thus, in general the workflow engine has to deal with a large number of intra-processor threads, each typically representing a Web Service invocation, which is likely to dominate the execution time. The emphasis is therefore on balancing the amount of intra- and inter-processore parallelism, while avoiding excessive load on the third-party services.

Inter-processor parallelism, which is available for processors that have no data dependency amongst them, is achieved by letting those processors begin execution in a new thread as soon as they receive their input data. The stack execution model also ensures intra-processor parallelism, by exploiting a model of *implicit iteration on collections*. The model is described extensively in [16]. Briefly, when a list value appears on a port where a simple type is expected, each element of the list is processed independently from the others[6]. This is a case of SPMD parallel processing [8], where the same function is applied concurrently to multiple data elements, and potentially results in multiple independent pipelines (see also the "Multiple instances with a priori runtime knowledge" pattern described in [17]), and is achieved in T2 by having the *Parallelise* layer allocate one thread to each element in the input collection, with an appropriate setting for the max number of threads. Since this layer is at the top of the stack, this has the effect of activating the processor on multiple concurrent requests (which may succeed or fail independently of one another). The collection of all the corresponding responses is then collated into a new output list, which is forwarded to the next processor. Since the requests may be served at different speeds, the elements of the response collection may be produced in arbitrary order. The *Parallelise* layer deals with this by simply waiting for the last element to arrive, before emitting the entire output list, with its order preserved. Additionally, however, a chain of processors which both iterate on their inputs, say P1 and P2, provide an opportunity for pipelining, as follows. The *Parallelise* layer of P1 forwards each response element *as soon as it arrives*, without waiting for the others and regardless of its position in the collection. In P2, each such element is again independent of the rest of the list, so the *Parallelise* layer of P2 can consume it immediately as part of a new thread. When extended to a chain of iterating processors, this strategy results in multiple, parallel pipelines, where both intra- and inter-processor parallelism is maximized. Clearly, any processor that requires to see the entire input before starting its processing represents a serialization point in the pipeline.

This form of *superscalar* and *streaming* pipelining [14] provides the basis for efficiently supporting workflow processing over streams of data, i.e., sequences of discrete input elements of unbounded length that are continuously produced by a source. Biomart[7][15] is an example of a service that supplies its output in a streamed fashion.

3   Performance Evaluation

In this section we compare T2's execution times and memory usage with those of T1.x, under a variety of experimental conditions. Focusing on intra- and inter-parallelism, we have program-

---

[6] The cited paper describes a much more general model that accounts for iterations on the cross product of multiple list-valued inputs.
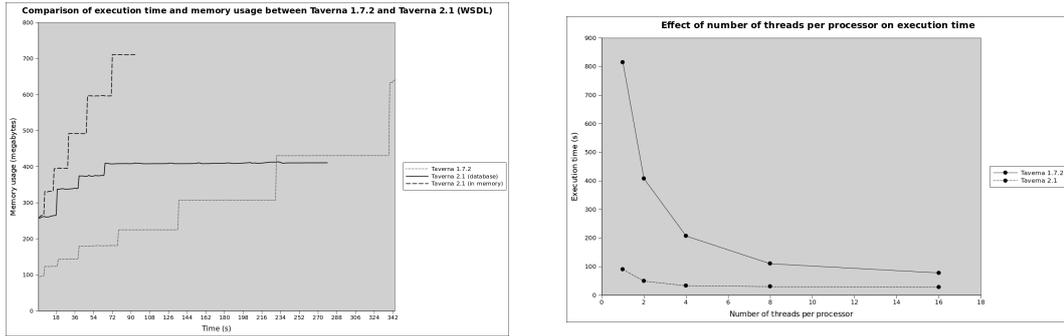
[7] `www.biomart.org`.

Fig. 3. Comparison of memory usage and execution times between T1.X and T2 with varying thread limits

matically generated a test workflow for performance analysis. The workflow consists of a linear chain of processors, each of which is made to iterate over elements of an input list of varying size. This simple workflow is sufficient to test the effect of both intra-processor parallelism, i.e., concurrent processing of the list elements, and pipelining through the chain of processors, as explained in Sec. 2.2. We have used this workflow to assess the execution times and memory usage for both T1.x and T2, with varying length of the input list and sizes of the list elements (strings). All experiments were conducted using a Taverna workbench running on a Java 6 JVM on a PC with 2GB of RAM and 2.3GhZ dual core processor. The workflows and source code used for all measurement are publicly available[8]. All processors invoke the same *echo* remote Web service, deployed on a concurrent server on a dual core machine on the same local network.

The experiments support the intuition that, when the workflow is structured in a way that makes pipelining available, T2 exploits it effectively, at the cost of an increase in memory usage, while T1.x must rely solely on intra-processor parallelism. Furthermore, the T2 Data Manager with a database back-end (as opposed to an in-memory data model) make the engine scalable over large data inputs. In the rest of the section we analyse these results in detail.

Fig. 3 compares the T1.x memory usage with that of T2 in two Data Manager configurations, namely (a) using a database (embedded Derby) or (b) in-memory data (in the latter, intermediate values are kept in memory throughout the entire execution). For these measurements, the workflow iterates over a list of 1,000 strings, each 10,000 characters in length, using 1 thread for each processor. The charts illustrate the trade-off between overall parallelism and memory usage. In particular, configuration T2(a) provides a "safe" option in that it guarantees bounded memory usage, at the cost of increased execution time over the faster T2(b). The shorter execution time of T2 over T1.x is due to pipelining, which is not available in T1. In this case, although each processor runs a single thread, each processor in the chain is activated as soon as the previous processor has produced *one element* of the output list. Thus, up to 10 concurrent threads exist in the system, each requiring a string to be loaded into memory. In T1, however, setting the max number of threads to 1 results in a serial computation through the entire chain.

Next, varying the max thread setting on each processor when comparing T1.x vs T2 reveals the impact of inter-processor parallelism, available only in T2, on overall memory usage and execution times. This is illustrated in Fig. 3, where the times are measured across different settings of the max number of threads. The plot for T2 suggests that, for this test workflow, this setting is not as critical as it is in T1.x. One reason is that, even with a small number of threads

---

[8] http://code.google.com/p/ws-menagerie/.

available for intra-processor parallelism, in T2 pipelining provides substantial inter-processor parallelism, resulting in lower overall execution times. A similar performance in T1.x requires 16 threads per processor or more.

Finally, Fig. 4 confirms that the T2 Data Manager ensures bounded memory usage that scales well with the size of the input and intermediate values. Not only does memory allocation stabilizes as the execution progresses, but, importantly, this is true across a range of data sizes that vary by an order of magnitude.

4   Case study: performance of a caGrid workflow

Taverna is the workflow model of choice for the caGrid project [18], which provides a service-based infrastructure consisting of data and computation resources designed to assist in-silico scientific investigation in cancer research [9]. As a specific case study, in this section we compare the performance of T2 against T1.x using a caGrid production workflow[9] used to carry out cancer diagnosis based on microarray analysis [11]. The workflow begins by extracting hybridization data, obtained from samples that belong to two different lymphoma types, from a microarray database. The data is then normalized and used to learn a classification model for lymphoma type prediction, using the Support Vector Machine (SVM) and K-Nearest Neighbour (KNN) algorithms. The model can then be used to classify lymphoma types from an unknown microarray dataset.

Our observations confirm the insight provided by the results presented in the previous section, regarding the time/memory trade-off available in T2. Specifically, Fig. 5 shows similar execution times (380sec. for T1.x and 450sec for T2 with a similar total number of threads, 40 vs 47), but better memory management for T2. The main difference between the two execution models is that T2 resolves references to microarray datasets, each about 10MB in size, on demand, transferring them from disk to process space and flushing them after use. This makes better use of memory but involves additional disk transfers. The in-memory model of T1.x saves transfer time but results in unbound memory usage.
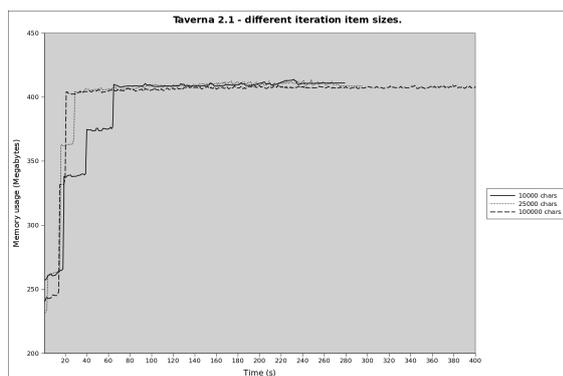

Fig. 4. Memory usage in T2 for different input strings lengths
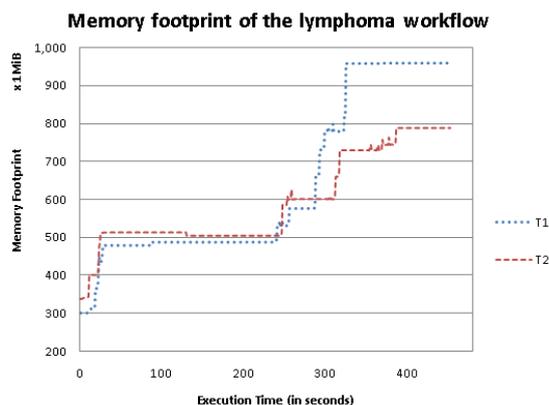

Fig. 5. Memory footprints for the lymphoma workflow in T1.X and T2

---

[9] The workflow, not reproduced here due to space constraints, is available from the myExperiment web site, at `http://www.myexperiment.org/workflows/746`.

As a second experiment aimed at showing the effect of pipelining, we have monitored the execution time and memory usage under different settings of max threads per processor, for a portion of the same caGrid workflow consisting of a linear chain of processors (Fig. 6), which are made to iterate over an input list of 10 experiment IDs. The results, shown in Fig. 7, indicate a near-linear correlation between max-thread setting on the processors and execution time, up to 10 threads per processor. A higher number of threads would bring diminishing returns, however, since the amount of real concurrency available on the Java-based workflow engine is limited by the number of cores that the JVM can use (2, in this experiment), and at the same time those threads saturate the concurrent server where the Web services execute, making it a bottleneck.
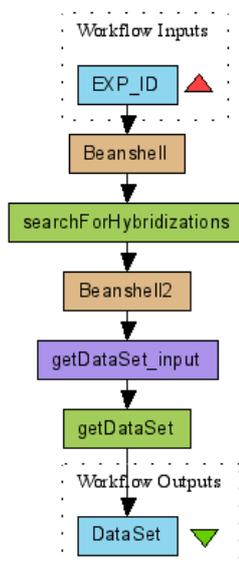
thread pool size	exec time (sec)	max memory (MB)	max thread count
1	41	317	47
2	29	355	49
5	24	383	55
10	21	442	56

Fig. 7. Execution times and memory usage by thread pool size

Fig. 6. Pipelined portion of the lymphoma workflow

5   Conclusions

We have presented the salient scalability and extensbility features of the Taverna 2 workflow management system, which make it suitable for data-intensive scientific applications. These features include, among others: (i) a runtime environment in which the available intra- and inter-processor parallelism that are implicit in the dataflow model are exposed as multiple execution threads, and (ii) extensibility and configurability points based on the interceptor pattern.

Our performance results, measured on a suite of programmatically generated workflows that exhibit both processor iteration and pipelining, indicate that T2 with RDBMS-based data storage offers good control of workflow execution memory while exhibiting competitive execution time.

Finally, we have presented a concrete case study that highlights the memory usage / execution time trade-offs on a production workflow from the caGrid project.

References

1. Peter Couvares, Tevfik Kosar, Alain Roy, Jeff Weber, and Kent Wenger. *Workflows for e-Science*, chapter Workflow M. Springer, 2007.
2. Ewa Deelman and Ann L Chervenak. Data Management Challenges of Data-Intensive Scientific Workflows. In *CCGRID*, pages 687–692, 2008.
3. Ewa Deelman, Gurmeet Singh, Mei-Hui Su, James Blythe, Yolanda Gil, Carl Kesselman, Gaurang Mehta, Karan Vahi, G Bruce Berriman, John Good, Anastasia C Laity, Joseph C Jacob, and Daniel S Katz. Pegasus: A framework for mapping complex scientific workflows onto distributed systems. *Scientific Programming*, 13(3):219–237, 2005.
4. P Fisher, C Hedeler, K Wolstencroft, H Hulme, H Noyes, S Kemp, R Stevens, and A Brass. A systematic strategy for large-scale analysis of genotype phenotype correlations: identification of candidate genes involved in {A}frican trypanosomiasisAfrican trypanosomiasis. *Nucleic Acids Research*, 35:5625–5633, August 2007.
5. Ian T Foster, Jens-S. Vöckler, Michael Wilde, and Yong Zhao. Chimera: AVirtual Data System for Representing, Querying, and Automating Data Derivation. In *SSDBM*, pages 37–46. IEEE Computer Society, 2002.
6. Y Gil, E Deelman, M Ellisman, T Fahringer, G Fox, D Gannon, C Goble, M Livny, L Moreau, and J Myers. Examining the Challenges of Scientific Workflows. *Computer*, 40:24–32, 2007.
7. D Hull, K Wolstencroft, R Stevens, C A Goble, M R Pocock, P Li, and T Oinn. Taverna: a tool for building and running workflows of services. *Nucleic Acids Research*, 34:729–732, 2006.
8. K Hwang and F A Briggs. *Computer architecture and parallel processing*. McGraw-Hill, New York, 1986.
9. S Joel, K Tahsin, H Shannon, L Stephen, and O Scott Et al. e-Science, caGrid, and Translational Biomedical Research. *Computer*, 41:58–66, 2008.
10. E A Lee. Dataflow Process Networks. Memorandum, UC Berkeley EECS Dept, 1994.
11. M. A. Shipp, K N Ross, P Tamayo, A P Weng, J L Kutok, and R C T Aguiar. Diffuse large B-cell lymphoma outcome prediction by gene-expression profiling and supervised machine learning. *Nature Medicine*, 8:68–74, 2002.
12. P. Missier, N. Paton, and K. Belhajjame. Fine-grained and efficient lineage querying of collection-based workflow provenance. In *Procs. EDBT*, Lausanne, Switzerland, 2010.
13. T Oinn, M Addis, J Ferris, D Marvin, M Senger, M Greenwood, T Carver, K Glover, M R Pocock, A Wipat, and P Li. Taverna: A tool for the composition and enactment of bioinformatics workflows. *Bioinformatics*, pages 3045–3054, November 2004.
14. C Pautasso and G Alonso. Parallel Computing Patterns for Grid Workflows. In *Proc. of the HPDC2006 Workshop on Workflows in Support of Large-Scale Science (WORKS06)*, Paris, France, 2006.
15. D Smedley, S Haider, B Ballester, R Holland, D London, G Thorisson, and A Kasprzyk. BioMart – biological queries made easy. *BMC Genomics*, 10, 2009.
16. D Turi, P Missier, D De Roure, C Goble, and T Oinn. Taverna Workflows: Syntax and Semantics. In *Proceedings of the 3rd e-Science conference*, Bangalore, India, December 2007.
17. Wil M P van der Aalst, Arthur H M ter Hofstede, Bartek Kiepuszewski, and Alistair P Barros. Workflow Patterns. *Distributed and Parallel Databases*, 14:5–51, 2003.
18. W. Tan I. Foster and R Madduri. Combining the Power of Taverna and caGrid: Scientific Workflows that Enable Web-Scale Collaboration. *IEEE Internet Computing*, 12:61–68, 2008.
19. Edward Walker, Weijia Xu, and Vinoth Chandar. Composing and executing parallel data-flow graphs with shell pipes. In *WORKS '09: Proceedings of the 4th Workshop on Workflows in Support of Large-Scale Science*, pages 1–10, New York, NY, USA, 2009. ACM.